DevOps Shack

Jenkins Part-3

Add Docker Container as a slave

```
pipeline {
  agent any
  stages {
    stage('Docker Build') {
             steps {
                script{
                    withDockerContainer(image: 'node:16-alpine', toolName:
'docker') {
                        //sh "git --version"
                        //sh "mvn -version"
                        sh "node --version"

}
                  }
               }
            }
        }
  }
```

Configure Mail Notifications

Steps in Video

```
pipeline {
    agent any

    tools {
        maven 'maven3'
    }

    stages {
```

```
        stage('Git') {
            steps {
                git branch: 'develop', url:
'https://github.com/jaiswaladi2468/BoardgameListingWebApp.git'
            }
        }

        stage('Build') {
            steps {
                sh "mvn package"
            }
        }
    }

    post {
    always {
        script {
            def jobName = env.JOB_NAME
            def buildNumber = env.BUILD_NUMBER
            def pipelineStatus = currentBuild.result ?: 'UNKNOWN'
            def bannerColor = pipelineStatus.toUpperCase() == 'SUCCESS' ?
'green' : 'red'

            def body = """
                <html>
                <body>
                <div style="border: 4px solid ${bannerColor}; padding:
10px;">
                <h2>${jobName} - Build ${buildNumber}</h2>
                <div style="background-color: ${bannerColor}; padding:
10px;">
                <h3 style="color: white;">Pipeline Status:
${pipelineStatus.toUpperCase()}</h3>
                </div>
                <p>Check the <a href="${BUILD_URL}">console output</a>.</p>
                </div>
                </body>
                </html>
            """

            emailext (
                subject: "${jobName} - Build ${buildNumber} -
${pipelineStatus.toUpperCase()}",
                body: body,
                to: 'xyz@gmail.com',
                from: 'jenkins@example.com',
                replyTo: 'jenkins@example.com',
                mimeType: 'text/html',

            )
        }
    }
}
}
```

Jenkins Shared library

Jenkins Shared Libraries allow you to define reusable code, functions, and steps that can be shared across multiple pipelines. This promotes code reuse, consistency, and maintainability in your Jenkins pipelines. Here's an explanation of the concept with an example using a Declarative Pipeline:

1. Create the Shared Library:

1.1. In your version control system (e.g., Git), create a repository for your shared library code.

1.2. Inside the repository, create a `vars` directory. This is where you'll define your reusable pipeline steps.
1.3. In the `vars` directory, create a Groovy file (e.g., `mySharedSteps.groovy`) for your shared steps.
1.4. Define the shared steps in the Groovy file. For example, let's create a simple step that echoes a message:

```
// vars/mySharedSteps.groovy
def echoMessage(message) {
    echo "Shared Library says: ${message}"
}
```

2. Configure Jenkins to Use the Shared Library:

2.1. In Jenkins, go to "Manage Jenkins" > "Configure System."

2.2. Under the "Global Pipeline Libraries" section, add a new library:

- Name: Enter a name for your library (e.g., `MySharedLibrary`).
- Default version: Specify a branch or tag in your repository.
- Retrieval method: Choose "Modern SCM" and select your version control system (e.g., Git).
- Project repository: Enter the URL of your shared library repository.

2.3. Save the configuration.

3. Using the Shared Library in a Declarative Pipeline:

3.1. In your project's Jenkinsfile, you can now use the shared steps defined in your library:

```
@Library('MySharedLibrary') _
pipeline {
    agent any

    stages {
        stage('Use Shared Steps') {
            steps {
                script {
                    echoMessage("Hello from Shared Library!")
                }
```

```
            }
        }
    }
}
```

3.2. The `@Library` annotation imports and uses the shared library in your pipeline.
3.3. The `echoMessage` step is provided by the shared library and can be used directly in your pipeline.

4. Benefits of Shared Libraries:

- **Code Reusability:** You can define complex logic, common patterns, and custom steps in the shared library and use them across multiple pipelines.
- **Consistency:** Shared libraries ensure that the same logic is applied consistently across different pipelines.
- **Maintainability:** Updates and improvements made to the shared library are automatically reflected in all pipelines that use it.
- **Versioning:** You can control which version of the shared library is used in your pipelines by specifying the library version in your Jenkinsfile.
- **Separation of Concerns:** By centralizing common functionality in the library, your pipeline definitions become more focused on the specific tasks of your project.

Shared Libraries are a powerful way to extend the capabilities of your Jenkins pipelines and promote best practices and standardization across your organization's CI/CD processes.

Skipping stages in jenkins pipeline

In a Jenkins Declarative Pipeline, you can use the `when` directive to conditionally skip stages based on certain conditions. The `when` directive allows you to define a condition using Groovy scripting, and if the condition is met, the specified stage will be executed; otherwise, it will be skipped. Here's an example of using the `when` directive to skip a stage in a pipeline:

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                // Checkout code from repository
                script {
                    checkout scm
                }
            }
        }

        stage('Build') {
            steps {
                // Build the project
                sh 'mvn clean package'
            }
        }
```

```
        stage('Deploy') {
            when {
                expression {
                    // Condition to skip the stage
                    return params.SKIP_DEPLOY == 'true'
                }
            }
            steps {
                // Deploy the application
                echo 'Deploying...'
            }
        }

        stage('Test') {
            steps {
                // Run tests
                sh 'mvn test'
            }
        }
    }
}
```

In this example:

- The `Deploy` stage is conditionally skipped based on the value of the `SKIP_DEPLOY` parameter.
- The `expression` block contains the Groovy condition that determines whether the stage should be skipped. In this case, if the `SKIP_DEPLOY` parameter is set to `'true'`, the `Deploy` stage will be skipped.
- The `when` directive allows you to control the execution of stages using a Groovy expression.

You can customize the condition in the `expression` block to suit your requirements. You can use environment variables, parameters, or any other variables accessible in the pipeline context to define your condition.
Keep in mind that the `when` directive is used to control the execution of stages, but it does not prevent the entire pipeline from running. Other stages will still be executed regardless of whether a specific stage is skipped.

Jenkins Backup

Taking backups of your Jenkins instance is crucial to ensure that you can recover your configuration, jobs, and data in case of system failures or other issues. Jenkins provides several ways to take backups, including manual and automated methods. Here's how you can take backups in Jenkins:

1. Manual Backup:

Backup Jobs:

- For Freestyle and Declarative Pipeline jobs, manually copy the job configurations from the Jenkins web interface.
- For Pipeline jobs defined in a Jenkinsfile, ensure your pipeline scripts are versioned in your version control system.

Backup Data Directory:

- The Jenkins data directory (often located at `/var/lib/jenkins` or a custom path) contains important files and configurations.
- Create a backup of this directory, including the jobs directory, plugins directory, and other configuration files.

2. Automated Backup:

ThinBackup Plugin:

- Install the "ThinBackup" plugin from the Jenkins Plugin Manager.
- Configure the plugin to schedule automated backups of your Jenkins instance.
- Backups can be stored locally, on a remote server, or in cloud storage.

Jenkins Backup to Amazon S3:

- If you're using Amazon Web Services (AWS), you can use the "Jenkins Backup to Amazon S3" plugin to automatically back up your Jenkins configuration and data to an S3 bucket.

Scripted Backup:

- You can write custom scripts to automate the backup process, including copying the data directory, job configurations, and other important files to a backup location.

3. Docker Backup:

If you're running Jenkins in a Docker container, consider using Docker-related backup mechanisms to create snapshots or export container data. Docker volumes can be backed up to ensure data persistence.

Important Considerations:

- Test your backup and restoration process in a non-production environment to ensure it works as expected.
- Store backups securely, preferably in an offsite location or cloud storage.
- Keep backup procedures and documentation up to date.
- Regularly review and update backup strategies based on changes in your Jenkins configuration and infrastructure.

Remember that backups are a critical part of disaster recovery planning, so ensure that you have a well-defined backup strategy that aligns with your organization's needs and requirements.

Troubleshooting

Troubleshooting in Jenkins involves identifying and resolving issues that can occur during the build, test, and deployment processes. Here are some common troubleshooting scenarios, along with examples and solutions:

1. Jenkins Job Fails to Start:

Scenario: A Jenkins job fails to start, and you're not sure why.

Solution:

1. Check the job's configuration for syntax errors or misconfigured settings.
2. Review the console output for error messages that indicate the cause of the failure.
3. Ensure that any required plugins are installed and up to date.

2. Build Errors:

Scenario: A build step within a Jenkins job fails.

Solution:

1. Review the console output for error messages or stack traces indicating the cause of the failure.
2. Verify that the build environment has the required tools and dependencies installed.
3. Check for issues related to source code, permissions, or file paths.

3. Job Stuck or Hanging:

Scenario: A Jenkins job appears to be stuck or hanging without making progress.

Solution:

1. Monitor the job's console output for any signs of activity or log messages.
2. Check if any resources (e.g., agents, external systems) required by the job are unavailable or experiencing issues.
3. Increase the timeout settings for build steps if applicable.

4. Git/SVN Checkout Failures:

Scenario: The job fails during the code checkout step from Git or SVN.

Solution:

1. Check the repository URL, credentials, and branch/tag settings in the job's configuration.
2. Verify that the Jenkins agent running the job has access to the Git/SVN repository.
3. Ensure that any required plugins for version control systems are installed and configured correctly.

5. Agent Connectivity Issues:

Scenario: The job fails due to connectivity issues with the Jenkins agent.

Solution:

1. Check the agent's status in the Jenkins dashboard.
2. Ensure that the agent's machine is reachable from the Jenkins master.
3. Verify that the agent's software and required tools are correctly installed and functioning.

6. Plugin Compatibility Problems:

Scenario: The job fails due to incompatibility issues with a plugin.

Solution:

1. Review the plugin versions and check if they're compatible with your Jenkins version.
2. Update the plugin to a version that's compatible with your Jenkins version.
3. Disable or remove plugins that are causing conflicts or compatibility issues.

7. Insufficient Disk Space:

Scenario: The job fails due to insufficient disk space on the Jenkins master or agent.

Solution:

1. Check disk usage on the machine hosting Jenkins.
2. Clean up unnecessary files or artifacts to free up space.
3. Consider increasing disk space or adding additional storage if required.

8. Configuration and Credential Issues:

Scenario: The job fails due to incorrect or missing configuration settings.

Solution:

1. Double-check the job's configuration for accuracy, including URLs, paths, and credentials.
2. Use Jenkins' credential management to securely store and provide credentials to jobs.

9. Network or Firewall Restrictions:

Scenario: The job fails due to network or firewall restrictions preventing communication.

Solution:

1. Verify that the Jenkins master and agents can communicate with each other and external resources.
2. Check firewall settings and ensure that required ports are open.

10. Plugin Update Issues:

Scenario: Updating a plugin causes problems in existing jobs.

Solution:

1. Before updating, review the plugin's release notes and documentation for compatibility considerations.
2. Test the plugin update in a non-production environment first to identify any issues.
3. If issues arise, consider rolling back the plugin version or seeking assistance from the plugin's community.

Remember that effective troubleshooting often involves a systematic approach of isolating and identifying the root cause of the issue. Check logs, console outputs, configurations, and relevant documentation to diagnose and resolve problems. If you're unable to solve a problem on your own, don't hesitate to seek assistance from the Jenkins community or your organization's support channels.